NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

HOW COGNITIVE PROCESSES AID
PROGRAM UNDERSTANDING

by

Paul Roderick Dorin

June 1985

Thesis Advisor: Gordon H. Bradley

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER	2. GOVT ACCESSION NO. / 3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)*: How Cognitive Processes Aid Program Understanding	5. TYPE OF REPORT & PERIOD COVERED: Master's Thesis, June 1985
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): Paul Roderick Dorin	8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Naval Postgraduate School, Monterey, CA 93943	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS: Naval Postgraduate School, Monterey, CA 93943	12. REPORT DATE: June 1985
	13. NUMBER OF PAGES: 63
14. MONITORING AGENCY NAME & ADDRESS*(if different from Controlling Office)*	15. SECURITY CLASS. *(of this report)*: UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for public release; distribution is unlimited

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

cognitive, software understanding, program understanding, chunking, slicing, hypothesis generation

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

A theoretical model of how an expert programmer goes about understanding a piece of software is presented. This understanding plays an especially critical role in software maintenance tasks. The model is based on three cognitive processes: CHUNKING, SLICING, and HYPOTHESIS GENERATION and VERIFICATION. These processes are used in conjunction with a programmer's knowledge base and categories (Continued)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

S N 0102-LF-014-6601

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

ABSTRACT (Continued)

of information critical to program understanding are identified. The model also takes advantage of certain characteristics of an associative memory to describe, using a semantic net representation, the mechanisms behind these processes and the organization of memory resulting from their use. The benefits of documentation and the use of commenting and mnemonics are described in terms of the model and may be useful as a guide for incorporating these into the code.

How Cognitive Processes Aid Program Understanding

by

Paul Roderick Dorin
Lieutenant Commander, United States Navy
B.S., United States Naval Academy, 1976

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 1985

ABSTRACT

A theoretical model of how an expert programmer goes about understanding a piece of software is presented. This understanding plays an especially critical role in software maintenance tasks. The model is based on three cognitive processes: CHUNKING, SLICING, and HYPOTHESIS GENERATION and VERIFICATION. These processes are used in conjunction with a programmer's knowledge base and categories of information critical to program understanding are identified. The model also takes advantage of certain characteristics of an associative memory to describe, using a semantic net representation, the mechanisms behind these processes and the organization of memory resulting from their use. The benefits of documentation and the use of commenting and mnemonics are described in terms of the model and may be useful as a guide for incorporating these into the code.

TABLE OF CONTENTS

LIST OF FIGURES

I. INTRODUCTION

Software maintenance now accounts for a large percentage of any software system's life-cycle cost. In view of this, the software industry has shifted its emphasis with respect to program evaluation. No longer is software being judged solely on the merits of its applicability to a given problem. While not neglecting the importance of this, the industry is considering factors which affect software maintenance as well. One such factor is software understandability [Ref. 1].

Gaining an understanding of unfamiliar programs is frequently cited by researchers as the first and often most costly step in software maintenance. This understanding is achieved when the programmer has 'learned' all that is necessary to competently carry out the required maintenance task. Making software easier to understand would have significant long term advantages resulting in reduced life-cycle costs. This study presents a theoretical model of cognitive processes, based on observed programmer behavior, which aids in acquiring this understanding. Further, the study contends that the effectiveness of these processes is dependent upon the extent of the programmer's knowledge base.

Most cognitive research analysing programmer behavior supports the idea of levels of skill or ability, and categorizes programmers as either novice, experienced, or expert. Based on the proposed theoretical model, this ability is defined by how well the processes are developed by the programmer, and the extent of his or her knowledge base.

A novice has a relatively limited knowledge base. Consequently, there is very little development of the cognitive processes in evidence. He or she is considered primarily a learner, using mainly unsophisticated techniques, such as inductive reasoning, to gain an understanding of a program.

An experienced programmer has a fairly extensive knowledge base. It includes information about most of the knowledge domains necessary for program understanding. The depth of information in these domains is, however, uneven. By this it is meant that an experienced programmer may know algorithms to perform a certain function, for example to sort numbers, but may find it difficult to adapt one of these to sort words. Or, in the category of programming languages, he or she may be familiar with the syntax and semantics, but unsure of the underlying design and its effects on a program.

Although still learning, the primary emphasis at this stage of a programmer's growth is the development of cognitive processes which make efficient use of this knowledge. At this stage, the programmer's performance is good, though inconsistent, over a spectrum of less difficult tasks. It does, however, degrade rapidly as task difficulty increases, indicative of only partially developed processes and the uneven knowledge base.

An expert, on the other hand, has acquired a broad knowledge base, including many specifics about programming languages and design, algorithms and data structures, task domains, etc., as well as how they relate to one another. He or she has a consistently high level of performance as well, proportional to task difficulty. This results from a demonstrated use of well developed cognitive processes.

These processes, which make use of the knowledge base, in conjunction with external information (program text, documentation, problem specifications, etc.), enhance the expert's ability to gain an in-depth understanding of the software involved in a given maintenance task. It is this demonstrated capability that distinguishes the expert from either a novice or experienced programmer.

Acknowledging this, the choice for this study is to model an expert involved in the task of understanding an unfamiliar program in order to perform some type of maintenance. What these processes are, how they are used,

and what information is contained in the knowledge base, form the major portions of this model. Realizing the subjective nature of the study, it is not a claim that this is a definitive model. It is, however, reasonable and representative of programmer behavior demonstrated by experts. In fact, this study contends that it is this very behavior of making efficient use of these processes which determines expertise in this area.

II. MEMORY and RECALL

We know empirically that information is remembered--stored in the brain--and can be recalled. Most evidence also supports the hypothesis that human memory is at least partly associative [Ref. 2]. By this it is meant that facts, events, concepts, and other types of information are encoded and stored in memory as separate elements or sets of elements, connected to one another by means of association. Each element is stored only once, but can have any number of associations with other elements. Each element is also directly accessible. One method of knowledge representation which incorporates many of the concepts and properties associated with this type of memory is the semantic net.

As there is no evidence that strongly supports any theory yet proposed to explain how memory and recall are accomplished, it should be noted that the model proposed here uses semantic nets only as a tool. The ideas of semantic nets will aid in explaining certain cognitive processes. However, the model itself has been developed based on research data and its validity is independent of this or any other theory regarding how these rudimentary cerebral functions, memory and recall, are accomplished.

Memory is commonly thought of as having two parts or areas. These are labeled Long Term Memory and Short Term

Memory. This may not be a physical division, though some researchers suggest that they're located in different areas of the brain, but rather one of cognition. Some researchers also include a third area, Working Memory. As the validity of this additional division of memory is not critical to the model, the simpler idea is adopted. A final form of 'memory', called External Memory, is also used.

A. SEMANTIC NETS

A semantic net is a directed graph made up of nodes, representing objects, connected to one another via links. These links indicate specific relationships or associations between nodes. This representation of knowledge is very popular among members of the Artificial Intelligence community. As there is no definitive set of characteristics for a semantic net, those relevant to the model proposed here are described. Much of this information is taken from a text by WINSTON [Ref. 3], whose description seems standard when compared to others in the literature. Properties have been added or altered, however, to aid in explaining certain behaviors of expert programmers. It is emphasized again that the model is based on observed behavior, and in no way depends on the validity of this presentation of semantic nets, or any other knowledge representation.

Three terms are used here to describe semantic nets. The objects of the net are called nodes and the relations

between objects are called links. They are represented in the figures by labeled circles and arrows respectively. A third term used by WINSTON, which is less standard, is the slot. The slots of a node are the different named links originating at the node. An example might serve here to better describe the use of these terms.

In Figure 1, we have an example of a semantic net. The five objects are CAR27 which is a specific car, CAR which is a general abstraction, DOUG and JILL which represent specific people, and the object BLUE. There is an OWNED-BY link between CAR27 and DOUG, and between CAR27 and JILL. There is an IS-A link between CAR27 and CAR, and there is a COLOR link between CAR27 and BLUE. CAR27 has four links associated with it, but only three slots. The COLOR slot is

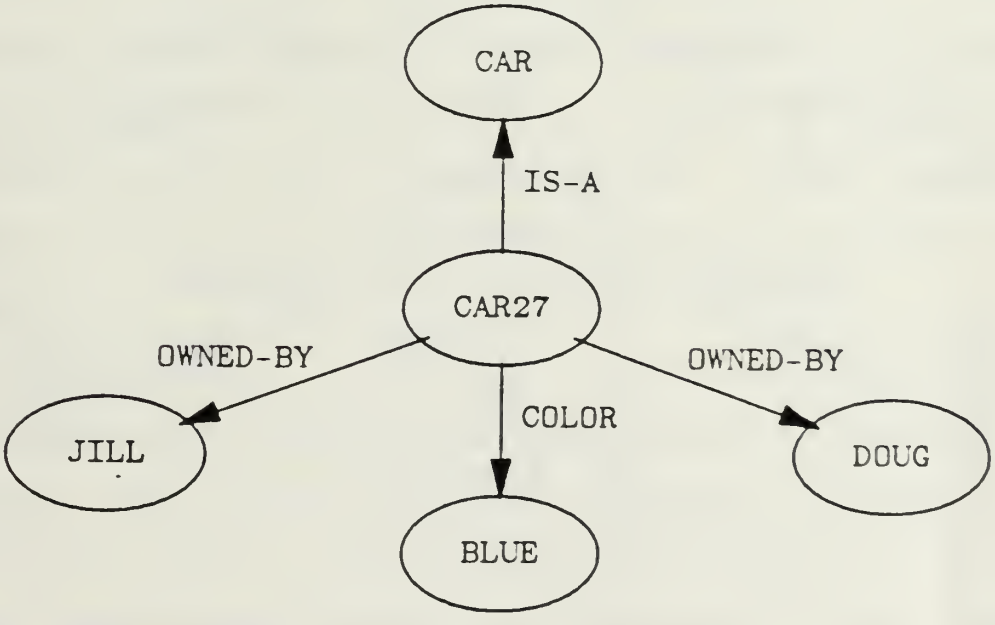

Figure 1 - A simple semantic net

filled with the value BLUE, the IS-A slot with the value CAR, and the OWNED-BY slot with the values DOUG and JILL. Note that the objects do not have to be tangible items, as illustrated by the object BLUE. Figure 1 is, of course, a representation of the knowledge that CAR27 is a blue car owned by Doug and Jill.

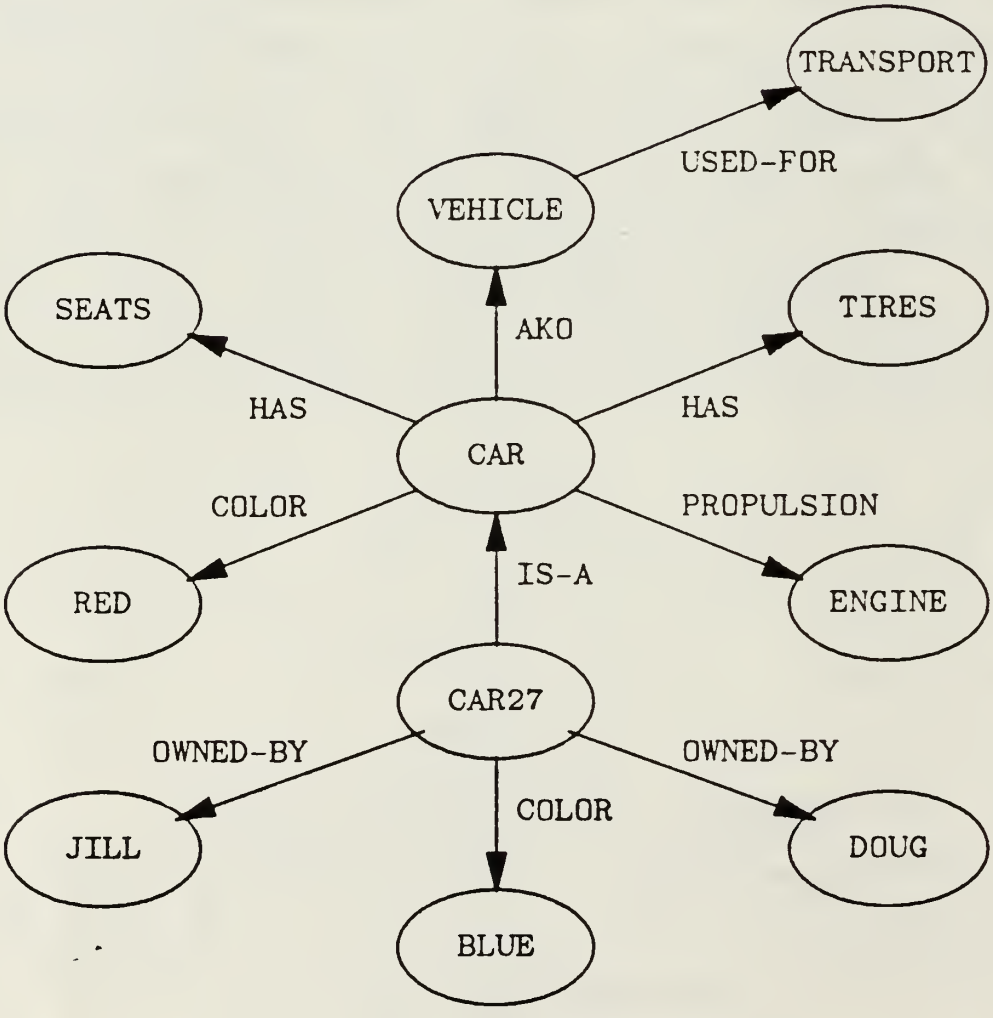

Figure 2 - Inheritance in Semantic Nets

When CAR27 is thought of, many facts about it come to mind. It has an engine, tires, and seats. Also, it is a vehicle used for transportation. Does this mean that, using our representation, the object CAR27 should have direct links to the objects ENGINE, TIRE, SEAT, VEHICLE, and TRANSPORTATION? The answer is no. The way this information is represented is through a property called inheritance, and the use of frames.

Inheritance is an object's acquisition of a slot value by inheriting the value from another object through association. Figure 2 is a semantic net showing one representation of the above facts about CAR27. As can be seen, CAR27 has no USED-FOR link, but does have an IS-A link to the more abstract object, CAR. However, it also has no USED-FOR link, but is associated to the object VEHICLE through an AKO - A Kind Of - link. In tracing the net from CAR27, VEHICLE is the first node reached which does have a USED-FOR slot value, TRANSPORTATION. CAR27, therefore, inherits this value through its indirect association with VEHICLE.

Again looking at Figure 2, notice the object CAR is linked to some familiar characteristics of a car via HAS links. This area of the net, isolated in Figure 3, is called a FRAME. A frame is a set or cluster of objects which serve as slot values for an abstract or less specific object. Its purpose is to group properties common to many

specific objects which are instances of the abstraction. These properties or slot values are then inherited by the more specific instances, making the net less complicated.

Slots can be added to or, although less likely, subtracted from a frame. This would occur due to additional information being incorporated into the net. Because of the dynamics of frames, they always represent the most current abstraction relative to the entire semantic net.

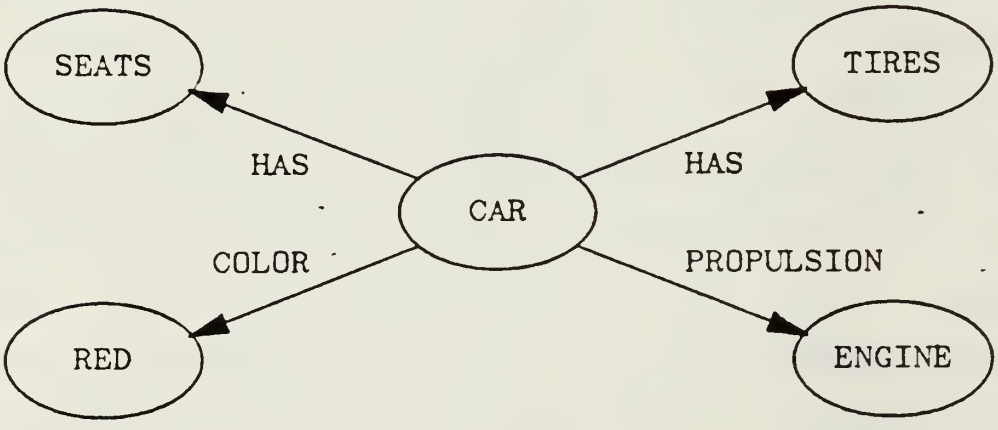

Figure 3 - A Frame

A frame also serves to provide DEFAULT values for incomplete pictures. Let's say, for illustrative purposes, that one of the slots of the frame representing CAR is the COLOR slot, and it is filled with the value RED. Now further suppose another object CAR2 is introduced, but without a COLOR link. Since all cars must have some

specific color, CAR28 is incomplete. To remedy this, it inherits the default value RED, until such time as its own color is added to the knowledge base.

Exceptions and unusual circumstances must also be accounted for. Using the CAR example again, suppose CAR28 is an experimental model using compressed air for power. The PROPULSION slot of the CAR frame is filled with the value ENGINE, yet for CAR28, this would be incorrect. Prior to knowing the method of PROPULSION, it is 'assumed' that CAR28 is powered by an engine. Once the method is known, however, a PROPULSION link is added to CAR28, reflecting the exception. Now, in trying to fill the PROPULSION slot for CAR28, the first value arrived at is COMPRESSED-AIR, the search stops, and the frame slot value becomes inconsequential. Figure 4 is the representative net.

By this explanation, it may appear that all objects making up a frame are default values, and exceptions nothing more than specific slot values in lieu of the default. Each, however, is subtly different. A frame is made up of attributes of an object. Some, such as engine, tire, or seat, are common to the majority and as such are not substitute values, used for lack of one more specific, but the same value shared among many objects. An exception is where particulars of an object contradict any of these shared values. Others, such as color, are common attributes with possibly different values for each instance of the item

whose abstraction is represented. These are truly default values, whose purpose is to fill a void until more specific information is obtained. This information is not an exception to the frame, but an expected piece of data previously missing or unknown.

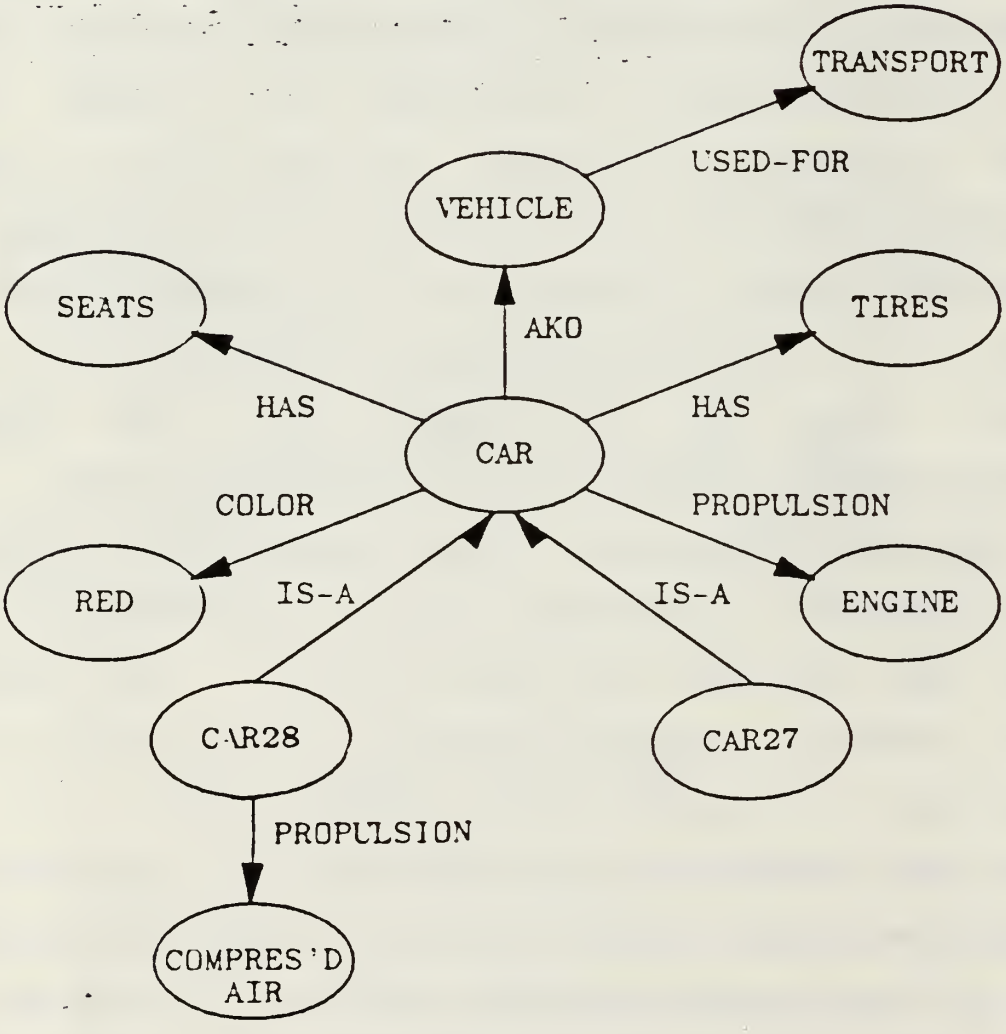

Figure 4 - Semantic Net with Exception

Another quality of an associative memory is the ability to distinguish the correct usage of an object, through context or perspective, when many different meanings exist. This dependency on context must also be represented in the net. Work cited by COHEN supports the idea that objects each have many classifications, determined by context [Ref. 4: pp. S-10]. This is because certain objects, when viewed from different perspectives, take on new or different qualities and attributes. A car, for example, can be looked at as an automobile, or as a toy, or as the car of a train. Obviously, each will have different attributes which are identified through context. The result is one object with three distinct purposes or aspects.

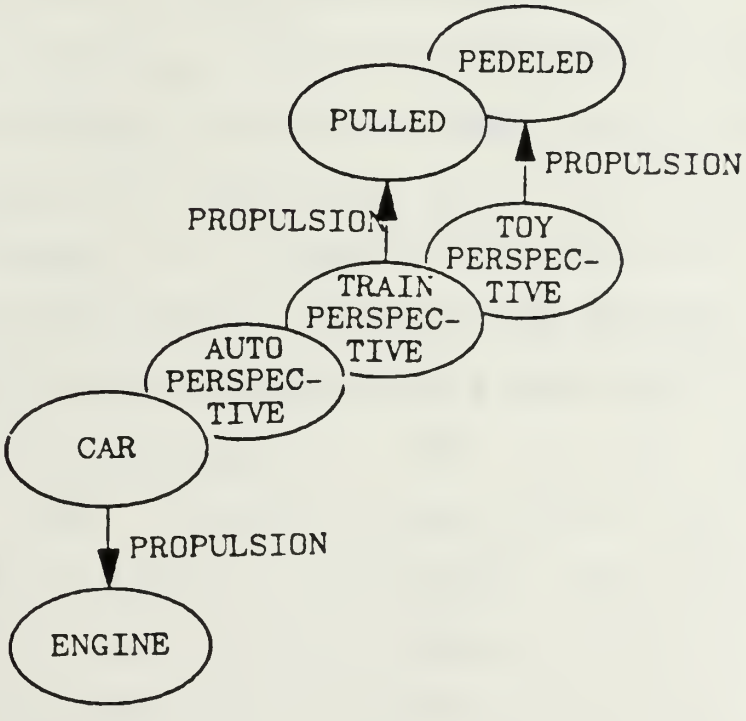

Figure 5 - A Perspective Node Bundle

One way to represent this in a semantic net is to view an object as a node bundle. This bundle consists of a general object node as well as a number of nodes each representing a different perspective for that object. Links relevant to a particular context are associated with the corresponding perspective node.

With such a representation, shown for CAR in Figure 5, slot values are accessed either with or without a perspective. Say, for example, the size of CAR is needed. If CAR is with reference to a train the returned value would be quite a bit different than if the inquiry were made for a toy car. If no perspective is given, the node bundle collapses to the single CAR node used throughout this example. This causes all possible slot values to be returned, each annotated with the associated perspective.

This notion of node bundles and object classification leads to the idea of node clustering. Put simply, a node cluster is a grouping in the net of objects and links strongly associated with one or two specific objects of the cluster. MINSKY uses a geographic analogy to illustrate the idea [Ref. 5: pg. 118]. He suggests picturing capitol cities with streets rowed by houses. These cities are connected via major throughfares to smaller suburban cities, which are in turn connected to towns, etc. The analogy to clusters, objects, and links is readily apparent.

The implication of this analogy is that semantic nets are organized hierarchically. If this idea is accepted, it follows that in order to recall a certain piece of information, several levels of the hierarchical structure must be transited depending on the point of entry. This walk through several levels necessarily has an adverse effect on the speed of recall. Yet, in some instances, information which should be separated by several levels is recalled faster than expected, implying an alternative method. To explain this, MINSKY introduces a second notion which allows for shortcuts through several levels. The argument is that if a certain path is reinforced a number of times through use, a direct link is formed, analogous to taking back roads to avoid lights and traffic.

These properties of semantic nets reflect those of an associative memory and will be referred to extensively throughout the remainder of this paper. Details will be added as necessary, to further explain behaviors, and this should make these semantic net properties clearer. However, it is important for the reader to understand these before proceeding.

B. SHORT TERM MEMORY

Information enters the cognitive system through short term memory. CURTIS [Ref. 6] quite adequately describes this memory as:

> "a limited capacity workspace which holds and processes those items of information currently under our attention."

This limited capacity was first quantified by MILLER as 7 ± 2 items [Ref. 7]. As will be seen later, an item is not limited to a single memory element, and may be a 'chunk' of indefinite size.

The information which exists in short term memory is transient and must be constantly used or 'rehearsed' to prevent its rapid decay [Ref. 8]. If the information is gained via perception, this rehearsal will, after a time, fix the information in long term memory. This is sometimes called the learning process. If, on the other hand, the information being used was recalled from long term memory, this rehearsal serves to reinforce it. This reinforcement has a positive effect on the future recall of this information and may cause it to migrate due to repetitive use. Both rapidity of recall and information migration are discussed later as they pertain to the model.

C. LONG TERM MEMORY

When we learn or memorize something, the information is retained in long term memory. When some event causes the recall of other events in the mind, the information comes from long term memory. It is the reservoir of permanent knowledge used in cognition, and has stored in it everything from the spatial model of the world to the motor and perceptual skills used moment to moment [Ref. 9: pg. 56]. Put simply, it is the knowledge base we operate from.

Unlike short term memory, the capacity of long term memory seems virtually unlimited. It receives and stores new information after processing in short term memory, and this information is directly accessible, once stored. Also, research has shown that the knowledge in long term memory is organized, and that the organization may change almost instantaneously, based on the context of the information being processed in short term memory. As will be seen later, this ability is significant in terms of the model, and will be discussed in more detail as it relates to an expert programmer's knowledge base.

D. EXTERNAL MEMORY

As an aid to information processing, external devices such as pencil and paper, chalkboards, and tape recorders are used to store information not in long term memory which the programmer wants readily available for reference. This

helps to compensate for the limited capacity of short term memory, and complements long term memory. All methods used for this purpose are generally referred to as external memories.

III. KNOWLEDGE BASE

> "Experts and novices differ in their abilities to process large amounts of meaningful information....A common explanation of this difference is that experts have not only more information, they have the information better organized...making their perception more efficient and their recall performance much higher." [Ref. 10]

The above quote emphasizes the importance of both the contents and the organization of the knowledge base. Included in the discussion presented here is the conviction that the contents of memory somehow affect this organization. Also, based on data from several studies referenced, this organization is dynamic and dependent on context.

A. CONTENTS

Along with basic knowledge, normally acquired through grade school and college, the expert programmer knows a great deal about five major categories of knowledge associated with programming. These are:

- ALGORITHMS
- PROGRAMMING LANGUAGES
- LOGIC
- DATA STRUCTURES
- PROGRAMMING DESIGN METHODOLOGIES

The depth of knowledge in these categories allows the expert to quickly focus on the important aspects of new information. Using the processes covered in the next chapter, he or she can then encode this information and relate it to what is already in long term memory.

Experts are familiar with many algorithms which do essentially the same job. Associated with each in the knowledge base is a set of benefits, drawbacks, applications, and, either implicitly or explicitly, a complexity evaluation. Choosing integer sorting as a representative task, there are several options: Merge Sort, Comparison Sort, Radix Sort, and Quick Sort to name a few. Each is useful in accomplishing the sort, however, each is also especially suited to certain applications. Each also has variations which are applicable to other types of sorts. The expert is familiar with these, as well as the underlying principles which differentiate them from one another. This allows him or her to readily adapt these algorithms to meet different needs, lexicographic sorting for instance.

Like algorithms, data structures have many variations. The expert is familiar with these and with the underlying principles behind their design as well. This allows easy modification to meet new requirements and aids the expert in recognizing design flaws such as lack of flexibility or expandability. The expert also has knowledge of algorithms and can correlate a given data structure with an algorithm

or group of algorithms for a specific application. The expert can also relate information on programming languages to data structures, evaluating the relative ease with which specific structures can be used and manipulated.

Programming languages are , to some degree, familiar to all programmers, whatever their skill level. An expert, however, is not only versed in the syntax and semantics of several languages. He or she is also familiar with the advantages and disadvantages of one language design, or particular machine implementation, over another. While the choice of language is not an option for the programmer tasked with maintaining or debugging, the particular design and implementation features play an important role when porting software from one machine to another.

Knowledge of language design and implementation also allows the expert to make judgements about software efficiency and memory needs. This knowledge also allows for identifying potential trouble spots, usually avoiding analysis of the entire program. This is particularly important when evaluating possible effects of a modification.

Information about algorithms also contributes to the knowledge of languages. As most languages have built-in functions, the expert can evaluate the particular algorithms used to implement these. This evaluation adds to his or her knowledge base of programming languages, aids in efficiency

analyses, and is useful in predicting the accuracy of results. Supported by this knowledge, an expert may choose to substitute other routines using more applicable algorithms, for such things as increased accuracy in calculations, more efficient device drivers, or faster access to secondary storage. He or she might also choose to replace programmed functions with ones built into the language, for the same reasons.

Knowledge regarding logic is important in two ways. First, it enables the expert to learn the specific implementation of control statements in a programming language, adding this to his or her knowledge base. Second, it aids in evaluating the flow of control in a given piece of software. Both help in analysing the efficiency of the software. Taking the following IF-THEN statement:

IF (A > 10) OR (B < 15) THEN C = D

the expert would know, or could test, whether or not the second comparison is executed independent of the result of the first. Taking advantage of this type of information could greatly impact the software's efficiency, saving money and CPU time.

Programming design methodologies are treated differently from other categories in the knowledge base. They can not be defined in specific terms, as we have done with the others, and are seen as more of a gestalt type of knowledge. They help the expert in analysing possible side effects,

which is, in part, a function of modularity. They play a major role in processes to be presented later, such as CHUNKING, SLICING, and HYPOTHESIZING.

Aside from knowledge of programming, the expert maintainer must also know something of the specific application area. The level or amount of information necessary is dependent upon the modification to be implemented. At the very least, however, the programmer needs to know enough to be able to interpret the documentation and program specifications in order to make a judgement regarding potential side effects of the change. This information is either learned information in long term memory, which can be recalled for future tasks, transient information used and then forgotten, or information kept as reference using an external memory.

The view of this study is that what is contained in the knowledge base directly affects the programmer's ability to understand a given piece of software. Obviously, what the programmer knows at the outset about the program's task domain, and information related to it, will impact on his or her difficulty in gaining this understanding. Extending this idea, a large disparity in the knowledge level significantly affects the level of competence of the programmer and, consequently, the relative quality of the work.

The cognitive processes which interact with this knowledge base, in order for the programmer to achieve this understanding, perform essentially three functions. Factual information is analysed and added to the knowledge base, or concepts and methodologies are abstracted from documentation, or information from one category is associated with that from another (such as correlating a data structure with an algorithm). These functions serve to integrate all information available to the programmer applicable to the task.

This knowledge base is not simply a collection of facts. It is the organized accumulation of information into a network reflecting semantic associations. This organization is equally as important as the information itself.

B. ORGANIZATION

Studies of recall show that people tend to organize information into categories and groupings. Most items or objects in memory are members of more than one of these categories, dependent on context. A piano is a member of the musical instrument category, and can be sub-categorized as a keyboard instrument in the context of musical instruments. It is also a member of the category which includes hutch and dresser when viewed as a heavy piece of furniture.

Grouping by order is another observed way memory has been organized. A person asked to list the ingredients of a recipe, for example, will more than likely list them in order of their use. When asked to list items necessary to equip a home, housewives listed these items either by category--kitchen utensils, furniture, window coverings--or by considering necessary items room by room [Ref. 4: pp. 8-11].

The evidence provided by these studies support the hypothesis that memory is organized dynamically, based on the context of the stimulus. It also implies that this organization makes use of information clustering. What is meant here is that information elements related by context 'migrate' toward certain key elements or toward one another. In either case, this clustering strengthens associations in context between these information elements, enhancing recall. As explained in a later chapter, this enhancement aids cognition by making pertinent information readily available to short term memory, while 'blocking' irrelevant associations involving these same elements.

Because these groupings are determined by context, the amount of information contained in the knowledge base associated with each element has a bearing on their categorization. The greater the amount of associated knowledge, the more refined the groupings can be. As more knowledge is gained and this refinement continues, new

clusters are formed to replace those less refined, and the association between any two becomes more specific. This, in turn, results in a reorganization of memory.

The studies cited here involve simple element lists. However, this idea is easily extended to more complex information elements, such as concepts, ideas, and abstractions, which are themselves clusters of information. The implication throughout this chapter is that different knowledge categories or domains are used best when integrated. How the contents and organization of memory relates specifically to the expert, and how this integration is accomplished, is addressed in the following chapter.

IV. THE PROCESSES

SCHNEIDERMAN and MAYER conjecture that, to facilitate program comprehension:

> "the programmer, with the aid of his or her syntactic knowledge of the language, constructs a multileveled internal semantic structure to represent the program." [Ref. 11]

The present study has identified, in the context of software maintenance, three major complementary cognitive processes, supported by certain lesser ones, used to accomplish this. Further, it is the tenet of the study that the entire program need not be represented in memory, but only that part which is of interest as determined by the programmer.

The descriptions of these processes have been formulated from observed programmer behavior. The ideas presented are extensions of theories based on empirical data resulting from limited testing. Introduction and subsequent treatment of these ideas in the literature has been, in many cases, artfully vague, with researchers characteristically relying on intuitive understanding through example. Therefore, although an attempt is made here to more clearly define these processes, the next chapter presents a scenario exemplifying the application of each.

A. CHUNKING

The cognitive process known as 'chunking' is a learned skill, enabling a programmer to encode information in such a way that a group of information elements can be represented and processed as a single element in short term memory [Ref. 7]. As mentioned previously, short term memory is where information processing occurs, and is characterized as having a limited capacity. This grouping or organizing of information allows programmers to operate on 'chunks' of associated information rather than single items. This translates to giving the programmer a broader perspective of the task.

Chunking is a very dynamic process, in terms of the knowledge base. A chunk is created when an association is formed between an encoded item in short term memory and its corresponding information cluster in long term memory. This cluster is the result of a reorganization of memory based on the context of the stimulus which initiated the chunking process. It can be added to or deleted from, based on the results of partial completion of the task for which it was created, or as information is learned, regarding the task, through other processes.

Chunking associations may also be formed between the encoded item and information in external memories. These associations may access information directly, or might simply guide the programmer to a reference in which the

necessary information is contained. In either case, they allow the programmer the use of transient or task specific information. At the same time, they alleviate the programmer of the burden of having to learn the information so it might be added to the cluster, or of having to store it in short term memory before it is needed.

The amount of information represented by a chunk is arbitrary [Ref. 12]. Its size is dependent on how much associated information is contained in the knowledge base, and to what extent external memories are used. The results of research by MILLER and others indicate that the number of items used or stored in short term memory is relatively constant. From this it can be concluded that the number of chunks which can be processed is independent of chunk size [Ref. 13: pg. 177, Ref. 9: pg. 44]. Thus, chunking effectively increases the capacity of short term memory as relates to information processing.

Besides having the ability to handle more information in short term memory, chunking also allows the programmer quick access to specific information which is part of the chunk. The reason is that chunks, representing information clusters, enhance recall of that information. All knowledge associated with the chunk has effectively been accessed, and can be thought of as staged for recall. This can best be explained by using a semantic net representation.

When the chunk is created, a reorganization of the knowledge base takes place, and information migrates, forming a high density node cluster. Again, the size of this cluster depends on the extent of the knowledge base. This density decreases the length of nodal links, resulting in a shorter walk from the initial access node or capital of the cluster to the desired information element. The association between the encoded item and the knowledge base is one example of the 'shortcut' described earlier, and links short term memory to the capital of the cluster.

The perspective has also been identified and associations between nodes not in context have been deemphasized. All the information represented by the chunk is now just beyond the programmer's consciousness waiting to be recalled. The encoded item can therefore be processed, representing a group of knowledge, with specific items associated with the chunk rapidly recalled for use when necessary.

Some researchers, such as KINTSCH, suggest that chunks, once formed, can be permanently stored in long term memory [Ref. 13: pg. 175]. This idea is inconsistent with the presentation here, and research for this study has uncovered no data to support the hypothesis. KINTSCH himself differentiates between what a chunk is in short and long term memory. His idea of stored chunks closely corresponds to the earlier presentation of information clustering. As

it is the contention of this study that a chunk exists only so long as it is under the programmer's attention, this notion of permanently stored chunks is disregarded.

B. SLICING

Expert programmers break large unfamiliar programs into smaller coherent pieces in order to gain an understanding of their function and/or design. Often, these pieces are determined by the original writers of the code. They are identified as blocks of code in the form of subroutines, procedures, functions, and the like. Identification is usually explicit and the pieces are written into the source as contiguous lines of program text. One can think of these as functional pieces of the program.

Also, experts routinely partition programs in ways that do not conform to textual, modular, or functional structure, permitting multiple views of the same code. Unlike functional pieces, which have a one-to-one correspondence between function and purpose of code lines, this type of division allows lines of code to be viewed from different perspectives. This associates a single line of code with more than one purpose. The construction of these views is what WEISER, who first proposed the idea, calls 'Program Slicing'. The process is used to strip from a program statements which do not influence a specific behavior or slicing criterion. The result is an abstract representation

of the program as viewed from the perspective of the specific behavior. This group of statements, usually associated with a single variable, is called a program slice [Ref. 14: pg. 439, Ref. 15: pg. 446].

Slicing is important in maintenance because typically only a subset of the program's behaviors is being improved or replaced. By eliminating non-influential code, the maintainer's job is made simpler. He or she can then deal with a much smaller 'program'. While this program may not be syntactically correct, it is semantically correct for the behavior of interest.

Also, the entire piece of software need not be sliced. If a point in the flow of control can be identified which bounds the slicing criterion, then only that part of the code still to be executed need be sliced. This further reduces the programmer's task.

Two key areas of the knowledge base are especially influential in determining the effectiveness of a programmer's slicing ability. Programming logic allows the maintainer to easily identify bounds of a specific behavior. He or she can, with an extensive knowledge base, trace through the program's flow of control easily and accurately, recognizing particular logic features of the language. Also, the expert's in-depth knowledge of the programming language gives him or her the ability to readily identify lines of code which impact the slicing criterion. For

example, familiarity with how data is passed and whether or not it is altered by code or simply used and returned without change (ie. Pass by Reference, Value, or Name) could greatly affect the size of the slice.

The extent to which experts employ slicing seems to depend on the program. Testing by WEISER shows that factors influencing the use of slicing are code size, structure, and ease of understanding [Ref. 15: pp. 459-461]. This suggests that slicing is found by experts to be most effective on poorly structured programs, and less so on those which are well designed and make use of modules, comments, and mnemonics. Effectiveness here is a relative measure of the amount of work eliminated and/or information gained by slicing.

The work by WEISER also demonstrates that expert programmers independently develop their own style of slicing. This does not preclude teaching its principles to less able programmers, but points out the process' dependence on the knowledge and experience of the individual. It also points to the fact that it is a subjective process and cannot presently be implemented fully. For the interested reader, however, WEISER does describe algorithms for approximating slices and discusses the effectiveness of two automatic slicing tools [Ref. 14].

C. HYPOTHESIS PROCESS

The third, and perhaps most powerful, process used by experts is hypothesis generation, refinement, and verification. It is a top-down process which allows for maximum utilization of the programmer's knowledge base, the overall depth of which determines the effectiveness of the process. It involves the generation, based on information in the knowledge base, and subsequent refinement and verification of hypotheses regarding the programmer's suppositions about how the code was designed and written. As more and more information about the software is processed, a hierarchy of these hypotheses is constructed.

This hierarchy is built quasi depth-first. This is because a programmer has a tendency to focus on one area, forming a cascade of refinement hypotheses through several levels before shifting his or her attention. The programmer does, however, remain cognizant of the other areas. Therefore, information encountered while refining the current area of interest is often used to form hypotheses relating to these other areas as well.

The hierarchical structure can be thought of as defining levels of understanding. The greater the depth, the more the programmer has refined his or her understanding of the software. By building this hierarchy, the programmer is creating an internal representation of the program, independent of any programming language. The goal or ideal

is that, at any level of understanding, the programmer should be able to produce a functionally equivalent program in any language that he or she is familiar with.

The title of the program, or a succinct presentation of the task for which the software was written, usually suggests enough information for the programmer to generate a hypothesis about the general flow of the program. This hypothesis would incorporate expected input and output types with a corresponding class or group of possible data structures. It would also have classes of algorithms and abstract logical constructs in its make-up, with the programmer essentially forming an overview of how the program might work. Note that these are classes and not specific elements.

As more information about the program is processed, these ideas are refined by generating other, more specific hypotheses based on new, more focused expectations. As mentioned, a hierarchy would begin to form, each level further refining the expectations used to generate the hypotheses above. As each new level is formed, it incorporates more information about the program. The result is more factual information in support of these hypotheses, and less supposition based on previous knowledge of similar tasks. This is not to say that knowledge base information is replaced by that newly learned about the task. Rather, facts about the problem are used to verify, whenever

possible, the supposed information. Only when a contradiction occurs is this information replaced. Obviously, this process is dependent on the programmer's having seen similar problems before. It seems appropriate, therefore, to digress for a moment to address this idea of sameness or analogy.

As was mentioned before, information in memory is organized into groups based on certain parameters or constraints. How, in fact, this grouping is accomplished, is still not understood, however it does occur. As associations are virtually limitless, it seems logical to assume that groupings are as well. Similar problems could therefore be grouped and an abstract set of circumstances formed to encompass dominant characteristics of the group. This idea is similar to that of a frame. Then, as problems are introduced, they are compared against these dominant characteristics. If the characteristics match, the problem is considered analogous.

As this matching process seems a mammoth task as presented, consider the reduction of work if these sets of circumstances were grouped by single characteristics, incorporating confidence levels, or another method of rating, to distinguish most from least dominant in the set. This would cause stronger and weaker associations, leading to the most probable set first, analogous to an electron following the path of least resistance. This type of

organization would greatly reduce the amount of searching necessary to identify this class of situations.

The benefits of these analogies, when they exist, are taken advantage of in generating hypotheses. As stated earlier, the programmer makes maximum use of his or her knowledge base. This is accomplished by relying on previously learned information regarding a general solution already familiar to him or her. In this case, the specifics of the software solution need only be learned if and when they are needed and differ from those of the general one. This is a much reduced task, relative to learning the entire solution (or program) when no such analogies exist in the knowledge base.

Returning to the discussion of hypotheses, the hierarchical structure can be explained easily by once again using a semantic net representation. Each hypothesis can be thought of as a frame. Each slot value of a frame would either be an information element or a frame itself, obviously more specific than the one whose slot it fills.

Initially, all frames (hypotheses) would contain either default or normal values. As more information is processed regarding the software, these values would be confirmed or replaced. These new values could be frames, representing still more specific hypotheses. Normal values, when contradicted, are replaced by exceptions specific to the problem at hand.

Each introduction of new information causes a reorganization of memory due to the change in context. This reorganization would make use of confirmed information, old or new, and may cause a change in default or normal values not yet verified. If this change in context occurs at a low level of the hierarchy, the programmer's perspective will change only slightly. If, however, the change affects slot values in the top levels, reorganization of a large subtree might occur, giving the programmer a significantly different view of the problem. The view could also change if the programmer chooses to shift his or her attention from the overall view, to a more refined hypothesis, focusing then on a subtree of the hierarchy. This would have the effect of emphasizing the details contained in this subtree and 'chunking' the remainder. The hypothesis hierarchy is therefore dynamic, changing with every shift in context.

Verification can take place at any time. It usually occurs when the programmer reaches a level of understanding about the behavior of the program that he or she wishes to confirm. This can be because the programmer has reached a level of understanding believed adequate for the task he or she needs to perform, or it might simply be to validate certain hypotheses before continuing. One reason for intermediate validation is that it lessens the effects of discovering an invalid hypothesis or contradiction.

For verification, the hypotheses forming the leaves of the tree are tested against the code. Two conditions are necessary for verification of the hierarchy. First, code corresponding to the hypothesis being verified must be in the program. Second, all code must be accounted for by one of the hypotheses. If either of these conditions fails, the structure is reorganized to reflect this and any new information gained from it.

V. SCENARIO

A scenario is now presented to help exemplify how each process applies to the task of program comprehension. It is meant to give the reader an intuitive understanding of application and effects, as well as the mechanisms underlying these cognitive processes. The reader should also gain an understanding of the interrelationships between the processes, the knowledge base, and information relating specifically to the program. It is the collective use of these which gives the expert his or her superior skills. For simplicity, a structured program is assumed as well as an ALGOL-like programming language. Again, semantic nets are used to represent memory organization.

The program used for this scenario will be one which computes averages of student grades and outputs a letter grade for each. It is a fairly structured program with adequate documentation and uses mnemonics but no comments in the source code.

A. A WALK-THROUGH

Suppose a programmer is given a program that he or she has never seen before and asked to perform some modification to it. Further suppose that to do this modification, an overall understanding of the program is necessary. He or she most likely begins by looking at the documentation.

After reading a small part of the documentation, perhaps a phrase or sentence, the programmer forms a hypothesis. He or she has ascertained that the program averages student grades. This defines a context, and a reorganization of memory takes place. This reorganization results in a large information cluster, forming a frame. It contains slots such as INPUT DATA, OUTPUT DATA, and PROCESSES.

The value of the INPUT DATA slot, based on the programmer's knowledge of how school grades are arrived at, is a cluster of possible types or classes of data. These would include, at this level, every type of data in his or her knowldege base that the programmer associates with school grades, as well as all possible data structures associated with them. The values of the other slots would be of a similar nature.

So by simply reading a single phrase, 'computes student grade averages', the programmer has constructed an internal representation of the program. He or she expects that it takes some input data, processes this data, and outputs the result. In addition, he or she has identified an input

domain, an output domain, and a domain of algorithms on which the processing of the data is assumed based. While this is certainly not specific enough a representation of the software to enable the programmer to do any useful work, a level of understanding has been achieved.

Further reading of the documentation reveals that each student's grades will be read in, summed, and the average converted to a letter grade and stored. This information suggests many, more specific, data and algorithmic classes, and several levels of hypotheses are formulated. Presuming that, at this point, the programmer begins to develop hypotheses in a quasi depth-first order, focusing on input, one hypothesis would be that grades are read in as numbers. Another might be that each student's identification is input in conjunction with his or her grades. The grade data hypothesis is then refined, forming a lower level hypothesis that grades will be represented as integers and handled as a list. Note that at this point, the programmer is not interested in what representation is used for student identification, possibly because hypotheses about the processing of the data suggest that the identification data will be used but not altered, so specific typing will not be necessary.

In memory, each hypothesis is represented as a frame with ordered slots. This ordering, if relevant, is based on the expected or confirmed ordering of the representative

information in the program, otherwise it is arbitrary. For example, the ordering of algorithms would be important in understanding the program, whereas the ordering of data classes in the frames created from the input hypotheses, for example the one representing the hypothesis that both grades and student identification are input, is not important for program understanding. If subsequent analysis reveals that a specific ordering is necessary, the frame would be reorganized to reflect this, because of the new context.

The value of each slot is an information cluster representing a knowledge domain, as frames representing hypotheses use classes of information and not specific elements. The cluster is formed based on the context defined by the hypothesis which the frame or slot represents. The initial hypothesis' INPUT slot has, as a value, a cluster representing all data types or classes that the programmer associates with grades. When the subsequent hypotheses are formed, defining the input as STUDENT IDENT and GRADE, this cluster is reorganized into a two slot frame, each representing a sub-cluster of the original. The value of the STUDENT IDENT slot becomes all possible representations by which students can be identified, and the value of the GRADE slot becomes the cluster of all possible classes of grade representation contained in the knowledge base. Any elements or nodes of the original cluster not associated with either of these new clusters is not

'visible' from this frame down, similar to the idea of scoping in some programming languages. So on one level, there is a single cluster representing the hypothesis as a grouping of all possible input data classes, while on another level, this same information, or a subset of it, is viewed as two separate clusters. This reorganization of information occurs because of the change in context when the subsidiary hypotheses are introduced.

The programmer has now increased his or her understanding of the program. In addition to what was expected based on the original hypothesis, the programmer now also expects that:

- grades are numerical
- each student's set of grades is processed separately
- the grades are initially input into a list structure
- the grades are summed and averaged
- each student is identified with his or her grades
- a mapping takes place from average to letter
- student ID and corresponding letter grade is stored

Figure 6 shows this representation focusing on the input subtree of the hypothesis hierarchy. Each level can be thought of as a level of understanding. It should be noted that, at this point, no verification has taken place and this level of understanding is contingent on the correctness

of the hypotheses formed. However, this understanding is not appreciably diminished unless the erroneous hypothesis is located in a top level of the hierarchy.

Continuing to focus on input, in order to verify this representation the programmer needs to slice the source code using input behavior as the criterion. Then, each line of code in the slice must be mapped to a leaf-frame or slot of the input subtree. Note that these leaf-frames or slots do not all have to be on the same level.

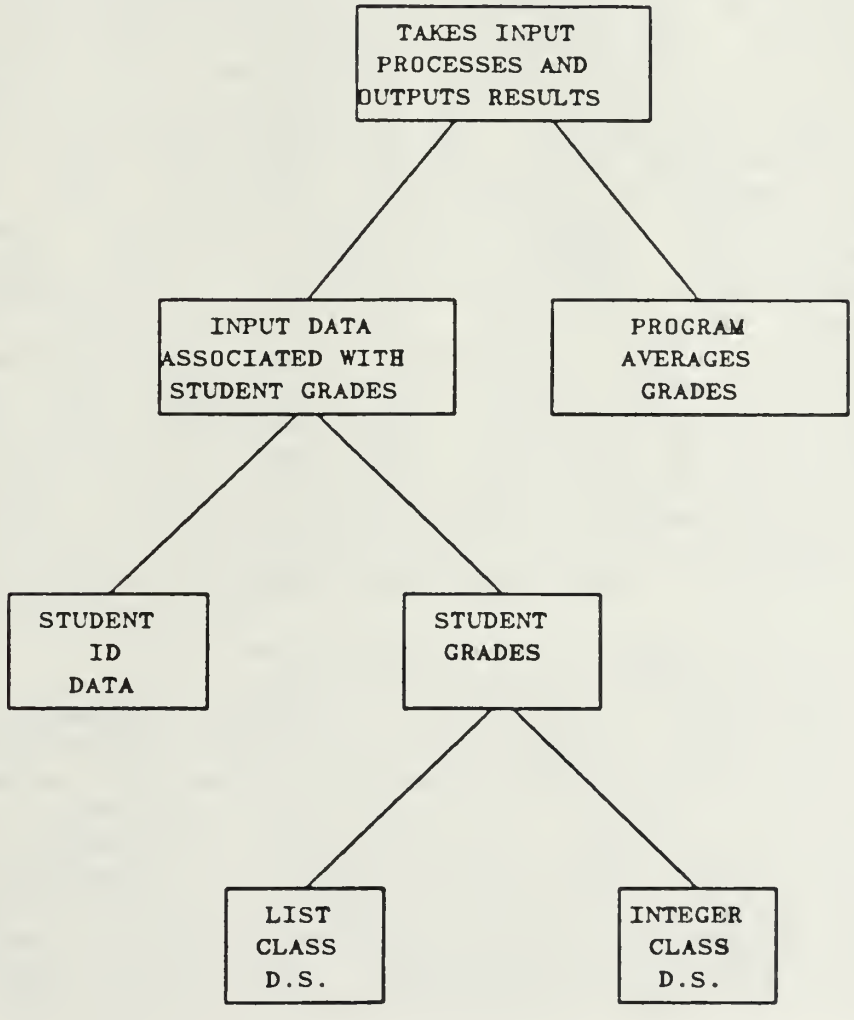

Figure 6 - Memory Representation of Program (Input)

Assume the following is the result of the slicing process:

```
READ STUDENT

REPEAT
  I = I + 1
  READ STUD_GRADE[I]
UNTIL STUD_GRADE[I] = 999
```

The programmer now attempts to verify the hypotheses against the code. The READ STUDENT line stands alone as verification of the hypothesis that each student is input. To verify the two hypotheses associated with grades is slightly more complicated. The READ STUD_GRADE[I] statement would be adequate to verify the hypothesis that student grades were input. However, it fails to confirm that it is a numerical representation. To confirm this, if no declaration statement exists, the programmer must analyse the behavior of the variable. The code resulting from the slicing process based on input is itself sliced, this time on STUD_GRADE[I]. The UNTIL STUD_GRADE[I] = 999 statement becomes the only other line in the slice.

The programmer recognizes the UNTIL statement as a compare and branch operation and notes that the variable is compared to a number. His or her knowledge of the programming language is extensive enough to realize that 999 must be a number and not a string. Also, he or she knows

that if a number is compared to anything but another number, a 'type mismatch' occurs. Therefore, STUD_GRADE[I] must be a number. This verifies the first slot of the frame.

The REPEAT-UNTIL block of the original slice is recognized as a looping construct. This, coupled with the fact that one variable inside the loop is used as an index, allows the programmer to chunk the block as "BUILD AN ARRAY". This chunk is associated with the grade input and, based on this context, the information cluster associated with the grade data structure is processed. It is found to include the class of array data structures, and so the second slot and its corresponding hypothesis is also verified. With all code now mapped, the entire input representation is considered verified, as all higher level hypotheses inherit the verification. Also, with reference to the last verification, it should be noted that the information cluster and hypothesis were further refined to reflect that a particular class, the array class, of list structures was used.

If a contradiction does occur in verification, a walk up the subtree takes place. Each hypothesis is checked until one is found which the information does not contradict. A new hypothesis is formed at the next lower level as a refinement of this hypothesis, and all hypotheses below this level are reevaluated based on the new context. A similar process takes place if information, other than that

expected, is found and needs to be included in the representation. Obviously, the higher up the tree the change takes place, the greater the memory reorganization necessary.

Up to this point, the programmer has been forming the program representation using a top-down approach. However, there are times when a bottom-up inductive approach is also necessary. Usually this approach is taken when a programmer's knowledge base, regarding the task domain, is incomplete, or when atypical algorithms are used. Here is where chunking plays a major role. The purpose of this next example is to demonstrate this role, and not to describe, in detail the inductive process.

Suppose the programmer is confronted with a module or block of code that he or she has formed no hypothesis about at a specific level. Using the grade averaging example, assume that the programmer has no knowledge of how averages are computed, and that the algorithm used is unknown to him or her. The programmer now tries to understand the algorithm by inductively reasoning about the code based on his or her knowledge of lower level functions performed within it.

At the lowest level, this is accomplished by looking at individual lines of code and assigning them interpretations [Ref. 12]. However, because the expert's knowledge base contains information about constructs and their uses,

certain of these lines are recognized as code included in the performance of a specific function. BROOKS calls these 'beacons'.

The block of code is for a standard averaging routine:

```
I = 1
SUM = 0

WHILE STUD_GRADE[I] < > 999 DO

  SUM = SUM + STUD_GRADE[I]
  I = I + 1

END_WHILE

AVERAGE = SUM / I
```

The programmer analysing this code recognizes the first two lines as assignment statements, and interprets them individually. He or she now looks at the WHILE line and recognizes it as a looping construct and beacon for several functional uses. The next assignment statement has the assignment variable on both sides of the equal sign, and so is interpreted as changing the value of SUM by performing some operation on it, rather than simply assigning it a value. Once the value added is recognized as an indexed value, the programmer chunks the loop. He or she has knowledge base information which shows that an indexed variable added to that type of assignment statement indicates an array summation process. So these four lines are chunked as "SUM STUDENT GRADES". Also, the first two lines are now chunked as "VARIABLE INITIALIZATION" based on

this new information. The last line is interpreted as an assignment statement which computes the grade average by dividing the sum of the grades by the number of grades summed.

By chunking, the programmer has taken a piece of code, which could be considered a single chunk which "COMPUTES GRADE AVERAGES", and formed a representation through inductive reasoning. The original seven lines of code can now be interpreted as:

- Initialize variables
- Sum grades
- Divide sum by number of grades summed

This representation can stay in short term memory to be used for the present task, being linked to the representation of the rest of the program in long term memory, and/or can be used to learn an averaging algorithm which could then be used for other tasks as well. And, once learned, the representation could be added to that in long term memory.

VI. RECOMMENDATIONS

This study has presented a theoretical model of simple cognitive processes developed and used by programmers. Further, the study has attempted to demonstrate how the expert, by using these processes, gains an in-depth understanding of complex programs. It is unrealistic, at present, to fully test these ideas because methodologies have not been developed in the behavioral sciences to do this. Also, the requisite size and complexity of the programs, and the time involved, are prohibitive. Research and the results of limited testing on small scale programs, however, do suggest certain design techniques, and coding and documentation methods which directly influence the effectiveness of these processes.

One such area is code structure, which should be designed so as to suggest chunks to anyone attempting to comprehend it [Ref. 13: pg. 175]. Functional elements of the code should be implemented as contiguous blocks of text whenever possible. Arbitrary GOTO's and forward and backward JUMPs should be avoided. Control flow statements should be used to direct flow from the exit point of one chunk to the entry point of others. All these considerations enhance the chunking process by making blocks

of code recognizable as single functions. This results in making it easier to use the text of the program as an external memory for those chunks.

Tests conducted by WEISER also indicated that code structure influences slicing [Ref. 15]. It was found that a much higher degree of slicing, among 21 expert programmers, took place when analysing a poorly structured program with indiscriminate use of GOTO's and non-mnemonic variable names than when analysing programs which make use of modular designs, mnemonics, and comments. The value of proper use of mnemonics and comments to the slicing process is that they serve to explicitly show data flow and to group associated statements and functions. This lessens the need for programmers to ferret out this information. One can conclude that less effort was required to achieve an equal level of understanding when good programming techniques were employed. The use of these maximizes the effectiveness of slicing while minimizing the effort necessary.

Comments and mnemonics are also helpful to the chunking process. A well placed comment, specifying the purpose of a block of code, and perhaps the data elements affected, explicitly identifies a functional chunk. This chunk could then easily be encoded based on the comment alone, eliminating the need for code analysis at that point. Meaningful mnemonics would give some insight into their purpose and thus both aid the recognition and chunking of

complex data structures and help to form correct hypotheses. These could then be incorporated into still larger chunks, allowing the many data elements which make up the structure to be processed as a single element in memory.

Program documentation can be, itself, a wealth of information for the expert programmer. A natural language explanation of the approach taken in originally designing the software facilitates the formulation of a fairly accurate hypothesis regarding its implementation. Citing explicitly the algorithms employed enables verification of certain hypotheses without extensive code analysis. Using this information, the maintainer can more easily focus on certain functions or behaviors of the code without having to first analyse it in depth to determine the specifics of its implementation. If exceptions to standard algorithmic coding are noted, it saves the programmer from having to determine why it was coded in such a way. Also, if subtle effects of the code are included in the documentation, along with certain potentials for side effects, it would reduce the testing necessary when a modification is made.

One final area which positively affects the use of these processes is standardization on all levels. Use of a standard design methodology would allow programmers to learn how to best chunk and slice certain representative software formats. 'Beacons' identifying certain functional areas

could be learned and used effectively. Automatic tools to aid these processes could also be developed with less difficulty.

On a more specific level, standardization of algorithms, and their corresponding constructs would greatly simplify the task of comprehension. Experts would be able to incorporate these into their knowldege bases, learning them from both the functional and the behavioral points of view. Also, coding templates could be learned and associated with these, aiding recognition of code itself.

Similar ideas have been used in most other engineering fields with great success. While software engineering is not, in many respects, as rigorous as these other disciplines, standards could be made flexible enough so as not to inhibit progress. Software reuseability is the motivation for recently generated interest in this area. The programming language ADA is the first step in an attempt at achieving some of this standardization, and its use in conjunction with these processes may serve to verify their validity.

LIST OF REFERENCES

1. Brooks, R., "Using a Behavioral Theory of Program Comprehension in Software Engineering," Proceedings of the 3rd International Conference on Software Engineering, pp. 196-201, IEEE Computer Society Press, 1978.

2. Wickelgren, W. A., Learning and Memory, pp. 11-23, Prentice Hall, 1977.

3. WInston, P. H., Artificial Intelligneee, 2nd ed., pp. 253-291, Addison-Wesley Publishing Company, 1984.

4. Cohen, G., The Psychology of Cognition, pp. 8-11, Academic Press, 1977.

5. Haugeland, J., Mind Design, pp. 118-120, MIT Press, 1981.

6. Curtis, B., "Fifteen Years of Psychology in Software Engineering: Individual Differences and Cognitive Science," Proceedings of the 7th International Conference on Software Engineering, pp. 97-106, IEEE Computer Society Press, 1984.

7. Miller, G. A., "The Magical Number Seven, Plus or Minus Two: Some Limits on our Capacity for Processing Information," The Psychological Review, v. 63, pp. 81-97, March 1956.

8. Tracz, W. J., "Computer Programming and the Human Thought Process," Software - Practice and Experience, v. 9, pp. 127-137, 1979.

9. Bower, G. H., and others, Handbook of Learning and Cognitive Processes, v. 1, Lawrence Erlbaum Associates, 1975.

10. McKeithen, K. B., Reitman, J. S., Rueter, H. H., and Hirtle, S. C., "Knowledge Organization and Skill Differences in Computer Programmers," Cognitive Psychology, v. 13, pp. 307-325, 1981.

11. Shneiderman, B., and Mayer, R., "Syntactic/Semantic Interactions in Programmer Behavior: A Model and Experimental Results," International Journal of Computer and Information Sciences, v. 8, pp. 219-238, 1979.

12. Brooks, R., "Toward a Theory of the Cognitive Processes in Computer Programming," International Journal of Man-Machine Studies, v. 9, pp. 737-751, 1977.

13. Kintsch, W., Learning, Memory, and Conceptual Processes, pp. 175-181, John Wiley & Sons, 1970.

14. Weiser, M., Proceedings of the Fifth International Conference on Software Engineering, pp. 439-449, IEEE Computer Society Press, 1981.

15. Weiser, M., "Programmers Use Slices When Debugging," Communications of the ACM, v. 25, pp. 446-452, July 1982.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defence Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93943-5100	2
3.	Professor G. H. Bradley Code 52Bz Naval Postgraduate School Monterey, California 93943-5100	4
4.	Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93943-5100	2
5.	Lieutenant Commander Paul R. Dorin 13343 La Venta Drive Poway, California 92064	4
6.	Doris Dorin 70 Toumaka Street Toms River, New Jersey 08757	1
7.	Mr. Paul Dorin 70 Toumaka Street Toms River, New Jersey 08757	1
8.	Lieutenant Commander Douglas L. Robbins 1 Surf Way Apt. 231 Monterey, California 93940	1